Describing and Using Object Frameworks

Hafedh Mili and Houari Sahraoui*
Département d'informatique, Université du Québec à Montréal
Case Postale 8888, Station Centre-Ville, Montréal (Québec) H2T 2T3
*Centre de Recherche Informatique de Montréal
1801 McGill College, Montréal (Québec) H3A 2M4
{mili,sahraoui}@larc.info.uqam.ca

Abstract

It has long been recognized that objects are too small units of reuse to provide any real leverage [Deutsch,1989]. When a set of objects are often used together to accomplish a frequently needed task, it is worthwhile packaging them as a unit; this is what we call object frameworks. Object frameworks are hard to design, to describe, to tailor, and to use. We set out to develop a representation of frameworks that satisfies the following, goals, (i) a description of inter-object behavior that supports both formal verification and understanding, (ii) abstraction and generalization, (iii) language-independence, (iv) an easy extraction of the descriptions from actual code, and more importantly, (v) support for reuse related tasks such as searchability, description of usage, incremental extension, and «operationalization»/realization. We propose a model that attempts to address these often conflicting goals, and that builds on existing work on both formal (e.g. [Goguen, 1986], [Mili, 1989], [Helm, 1990], [Mili, 1996]) and informal (see e.g. [Johnson,1992]) inter-object behavioral description techniques. Our model supports abstraction through a set of closure operations that allow us to treat framework descriptions as regular class interfaces, supporting the embedding of frameworks. We define a packaging operation on frameworks that packages a specific instantiation of a framework into a concrete class definition, which may be used as a concrete participant in yet more aggregate frameworks, and which supports various optimizations. We describe an ongoing prototype framework definition and manipulation tool set that supports C++ frameworks, which we integrated into a toolkit for searching and browsing reusable components. We conclude by discussing future developments for the prototype and directions for future research.

1. Introduction

Reusability is often cited as the major contribution of object-orientation to the engineering of software. Since the late 80's, it became clear that classes are too small units of reuse to realize the projected or promised benefits [Deutsch, 1989]. This is due to two related factors. First, most classes tend to be rather small in size, and second, classes seldom perform useful behavior by themselves; they often *contribute* to a given function by interacting with other objects [Mili et al., 1990]. This led researchers and practitioners alike to look into principles for designing and reusing collections of reusable interacting classes that, together, perform some useful functions (see e.g. [Johnson & Foote, 1988],[Deutsch, 1989]). We refer to such collections of objects as *object frameworks*. Object frameworks can be of two kinds, *domain-specific frameworks*, with consist of a set of classes from a specific application (business) domain, and *design frameworks* which consist of

classes that implement a given *mechanism* or a *computer-related task*. We are interested in both kinds of frameworks, and in this paper, we propose a representation and search model that should accommodate both, and describe a preliminary Smalltalk prototype.

Developing and reusing object frameworks poses many challenges. From a development point of view, there are two competing goals. Before we invest into implementing a framework, we have to make sure that the design or domain it embodies is general enough that it will often be useful. At the same time, it must be precise enough to be programmable and to have a reasonable performance. This relies on a number of parameterization techniques that enable us to develop as much of framework's generic functionality as possible, leaving only application- or usage-specific behavior to be developed. Such techniques include general object-oriented abstraction techniques such as abstract classes, which play the role of place holders to be replaced by actual application classes, or generic classes which may be parameterized by such classes, or framework specific techniques such as interface bridges (e.g. the so-called pluggable adaptors in Smalltalk's MVC), or the broadcast-based management of inter-object communication (e.g. Smalltalk's dependents mechanism). From a reuse viewpoint, there are two challenges. First, we have to find a way of documenting frameworks succinctly, unambiguously, understandably, and in a way that can be matched against developers' needs and that can allow them to use it. These are conflicting needs and neither formal specifications (see e.g. [Helm et al., 1990], [Hall & Weedon, 1993]) nor the so-called pattern languages (see e.g. [Johnson, 1994],[Johnson, 1994],[Gabriel,1994]) address all of them. Second, there is the problem of searching for relevant frameworks in the first place, and once one is found, searching for classes that support a given interface, or ascertaining that a given class does. In this paper, we propose a model for describing object frameworks that attempts to address these issues.

In section 2, we provide a more thorough discussion of the challenges described earlier in the form of a set of requirements that our model must satisfy. In section 3, we illustrate the model through a simple example involving two C++ frameworks dealing with event-based simulation systems, and describe a preliminary Smalltalk implementation. In section 4, we discuss related work. We conclude in section 5.

2. Requirements

In this section, we discuss the considerations that motivated us to choose the model described in section 3. This concerns both the set of descriptive elements that our model must support, as well as the form that these elements must take.

2.1. Backward applicability

Our purpose is as much to represent existing object frameworks in a way that makes them more useful and more usable, as it is to propose a methodology for *building* an object framework, with a focus on detailed design and packaging issues. This means that we will be seeking a trade-off between a set of constructs that can be easily mapped to existing frameworks, despite their inadequate documentation or less than perfect packaging (from an abstraction point of view) and a set

of constructs that represent all that we would ever want to know about an object framework, and that rely on the use of novel abstraction techniques. We will see examples of these trade-offs throughout this section and the next.

2.2. Representing inter-object behavior

A framework involves the collaboration of several objects. Understanding what a framework does and how it does it requires a description of the interactions between objects, or *inter-object behavior*. Representing inter-object behavior poses a number of challenges, including: 1) choosing between representing the computational versus functional properties of the system, and 2) choosing a trade-off between flexibility and abstraction, on the one hand, and ease of verification, on the other. We discuss the two sets of issues below.

Computational vs. functional aspects: The behavior on an object or system of objects may be described in terms of *computations* or in terms of *functions*[1]. The computational description tends to describe the behavior in terms of abstract, application-independent computational devices or application-independent transformations of data, regardless of their semantics. By contrast, a *functional* description describes inter-object behavior in terms of application meaningful and purposeful behavior. Computational (abstract) descriptions are useful for verification and validation purposes, and to ascertain the conformance of a given class/implementation to a desired behavior, provided that both are described computationally. However, they do not promote understandability and do not convey, in an application-meaningful way, the *purpose* of the behavior [Mili et al., 1995] or relate it to developers/reusers' needs.

The distinction between functional and computational aspects becomes fuzzier with design patterns, and reusable designs in general, which are not distinguishable by what they do, but rather by optimizing different *design quality criteria.*

Flexibility vs. ease of verification: One of the purposes of behavioral descriptions is to compare requirements on framework participants, to actual specifications of candidate components. When choosing a behavior representation language, we have to make sure that it is abstract enough to support flexible matching between requirements and specifications that is independent of inessential implementation details. However, we also have to consider how to derive such specifications for existing frameworks or for frameworks yet to be designed. Practically, it would be helpful to have a simple relationship between such specifications and the actual code so that the specifications can be extracted, in part or in full, from the source code.

In existing frameworks, inter-object behavior is typically embodied in explicit cross-references (calls) between object methods. Using such cross-references as specification for inter-object behavior has two inherent problems. First, it dictates one mode of interactions between methods, as opposed to another. For example, two processes or objects can communicate either through interruptions or through explicit calls; our choice for showing inter-object linkage should be, to the extent that that is possible, mechanism-independent. Second, if we use explicit cross-refer-

1. The philosophy of science makes one such distinction between *behavior* (which we call computation) and *function*. *Behavior* is a simple input-output relationship, while *function* is behavior within the context of a an englobing behavior [Darden & rada, 1988]

ences, we impose a non-essential *lexical coupling* between methods and objects which makes the inter-change of candidate objects very difficult. This is as much a problem with the detailed design and implementation of the framework as it is a problem with describing it.

2.3. Describing usage

Describing the usage for a framework, or any other reusable component, involves two issues. First, there is the issue of appropriateness of the framework to a given need or situation. Second, there is of issue of describing *how to use* the framework. The issue of appropriateness or *opportunity* is related to the description of inter-object behavior mentioned earlier. With application-specific frameworks, both the framework and developers' needs will be expressed in the application domain language. With design patterns, we need a computational abstraction of what the pattern does, *and* a description of which design criterion it maximizes. This is the approach followed by the design patterns community where textual documentation, albeit informal, is carefully structured so as to reflect these distinctions (see e.g. [Johnson,1992], [Gamma et al.,1995]).

With regard to the *how to use* aspects, in addition to the traditional information that a reuser needs to know, such as the acceptable types of parameters, any side effects, object frameworks have the distinction of requiring assembly. This may involve three things: 1) selecting participants, 2) preparing participants so that they can interact with the other components of the framework, e.g. by adding bridge methods, and 3) instance creation and linkage. The selection is usually based on the role played by the participant. The preparation is often required because the *actual* participant that plays the right role may not have the proper interface.

2.4. Scalability and extensibility

Scalability means that we should be able to build and describe complex frameworks by assembling simpler ones, or, equivalently, we should allow the participant of a framework to be another framework. Extensibility means that we should be able to incrementally augment or specialize the functionality of a framework. These requirements imply that a framework be packaged as a unit with its own external interface and its internal structure, similar to the way classes are packaged.

Finding the "external interface" of a set of interacting objects requires a *closure* operation on the set of messages that they exchange within each interaction sequence so that, from the outside, such a sequence may be assigned a method-like signature. This is difficult, both computationally, and in terms of ascribing a useful meaning to the resulting signature-- not a problem with class methods which tend to be cohesive. Another difference between class packaging and framework packaging has to do with the information hiding that typically characterizes their respective boundaries (see e.g. [Wegner, 1992]). While a class may be seen and reused as a black-box, to use a framework, we need to have access to its participants. At best, a framework may be seen as parameterized (generic) black-box.

2.5. Searchability

We envision a two-stage search. In the first stage, a developer specifies a need, and we have to

find the reusable component that best satisfies that need. Depending on the need and how it is expressed, the answer could be a method, a class, or an object framework. If the answer is an object framework, we then have to search for suitable participants, or validate candidate ones.

The first stage of the search raises a number of issues related to the issues of closure, unit packaging, and to the issue of computational versus functional descriptions. First, a developer's need will typically be expressed in a synthetic fashion without referring to a specific architecture or to a distribution of responsibilities between interacting objects, compelling us to find a synthetic way of expressing the functionality of a framework that abstracts the specific participants. Second, a functional need will often be expressed in terms of application semantics, which has to be matched to whichever description is available for the functionality of the framework. Consider the query "I need a way that allow me to represent dynamically the load of the CPU using a bar chart". If we had an application framework in the library which deals with processes, we may find a single class, say **ProcessMonitor**, which does (encapsulates) just that. If not, we need to translate the query into application-independent, programmatic terms, such as "I need a way that allows me to represent dynamically the <attribute> of <an object> using <a graphical object>". Such a query is "aware" of the distinction between a model object and a graphical object, and might retrieve the MVC framework, if it were available. Yet a more abstract formulation might say "I need a mechanism for propagating state changes between related objects", which would retrieve Smalltalk's *dependents* mechanism, which is used, among other places, in the MVC. Developers need to express their needs in whichever language can be matched to available, or automatically derivable, descriptions for frameworks. Looking at the problem another way, the above three formulations could also be successive translations by a search engine as it fails to find adequate matches. They can also describe nodes along a generalization or extension path of frameworks, moving from the {**Process**, **ProcessMonitor**} framework, to the {**Model**, **View**, **Controller**} framework, to the {**Object**, **Dependent**} framework[1].

The second phase of the search involves interface matching between framework participants and concrete classes. Our major concern is that such a matching be workable on existing class libraries, without requiring extensive manual packaging.

3. Model and implementation

In this section, we describe our model for representing object frameworks and the prototype implementation that we are currently developing. Our presentation will focus on those aspects which are inter-dependents and which involve some of the trade-offs discussed above. First, we motivate the major constructs used to represent object frameworks through two examples. In section 3.2, we summarize the representation model. In section 3.3, we discuss our approach to the unit packaging of object frameworks. We conclude in section 3.4 with some implementation notes.

1. It should be {**Object**,**Object**}, but for readability purposes, and because framework participants would refer to interfaces rather than classes, we use this characterization.

3.1. Example

We take the example of two frameworks that are part of the OSE Library [Dumpleton,1994], which are aimed at building process simulation systems (OTC_Simulation_Systems), and event-based systems (OTC_Event_Based_Systems), respectively. The frameworks weren't referred to as such in [Dumpleton,1994], but the -- unusually high-quality-- documentation described the component classes together. We use these examples to discuss the descriptive elements of the framework, and address some of the packaging issues raised in section 2.

3.1.1. Describing what a framework does

In process simulation systems, there is a queue of jobs to be executed. A dispatcher retrieves the first job from the queue and executes the code associated with it. The OTC_Simulation_Systems framework offers two generic classes, **OTC_Job** and **OTC_Dispacher**. A potential instance of this framework must include one or more instances of-- possibly different- subclasses of **OTC_Job**, and a single instance of a class that specializes **OTC_Dispacher**. We call these potential instances the **participants** of the framework. Each of these participants has to satisfy a predefined interface. For example, a job must understand the messages `execute()`, `destroy()`, etc. We show below what the specification of the participants and their interfaces looks like. We will be using what amounts to a C++-like notation, with some syntactic sugaring, to describe framework specifications and actual implementations.

```
Framework OTC_Simulation_System
{
Participants :
     Interface : Dispatcher {
          Attributes :
                    jobQueue Queue;
          Signatures :
                    void initialise();
                    void initialise(IN queue Queue);
                    void schedule(IN job Job);
                    Integer run();
                    Integer dispatch()
          ...
          }
     Interface : Job {
          Signatures :
                    void execute();
                    void destroy();
          ...
          }
     Interface : OTC_Queue {
          Attributes :
                    jobs List<Job>;
          Signatures :
                    void add( IN job Job);
                    Job next();
          ...
```

```
            }
        d Dispatcher;
        j* Job;
        q Queue
...
}
```

Notice that both `Dispatcher` and `Job` are names of *interfaces*, which may or may not correspond to actual classes. In practice, however, interfaces are typically represented by *abstract classes* and to make sure that candidate participant classes support such interfaces, we require that they be subclasses of the abstract class. The next figure shows one such class, `Count_Job.` The class `Count_Job` provides implementations for the methods required in the interface, and adds its own methods and instance variables.

```
Class Count_Job : public OTC_Job {
public:
     Count_Job()
     :runnable(OTCLIB_TRUE) {};
     void kill () { runnable = OTCLIB_FALSE;}
     void execute();
     void destroy();
private:
     static int count;
     OTC_Boolean runnable;
}
```

The interaction between the participants of the framework may be described using the following cycle:

- The dispatcher receives the message `run()`,
- The dispatcher sends the message `dispatch()`to itself,
- The dispatcher sends the message `next()`to the queue, which returns some job jb,
- The dispatcher sends the message `execute()` to *jb*,
- When the *jb* terminates, the dispatcher sends it the message `destroy()`.

There are different ways of representing this interaction. A plausible C++ implementation might look like the following:

```
void Dispatcher::run() {
     Int result = 1;
     while (result > 0) result = dispatch();
     return result;
}
Int Dispatcher::dispatch() {
     Job * jb;
     jb = jobQueue -> next();
     if (jb == 0) return 0;
     jb -> execute();
     jb -> destroy();
     return 1;
}
```

In this implementation, the message `dispatch()` refers *explicitly* to the messages of the other participants that are involved in this sequence. Such a practice is to be discouraged both for describing inter-object behavior and for implementing it, as mentioned in section 2.2. Our

approach for representing this behavior is to use a generic message connection notation which can be easily mapped to a variety of message connection mechanisms and control paradigms. For the time being, we use a single connector denoted by $\Rightarrow$. The symbol $\leftarrow$ is used for assignment.

$$d \cdot run(\,) \Rightarrow d \cdot dispatch(\,) \Rightarrow \begin{bmatrix} i \leftarrow q \cdot next(\,) \\ i \cdot execute(\,) \\ i \cdot destroy(\,) \end{bmatrix}$$

The message sequence shown above corresponds to a unit of behavior that can be performed by the framework. For scalability purposes, we need to package this unit as a single message (see section 2.). This requires a closure operation on message sequences that assigns to a message sequence a single message that has the same *external* signature as the original sequence. Figure 1 shows a message flow graph where a boundary is drawn around the components of a framework.

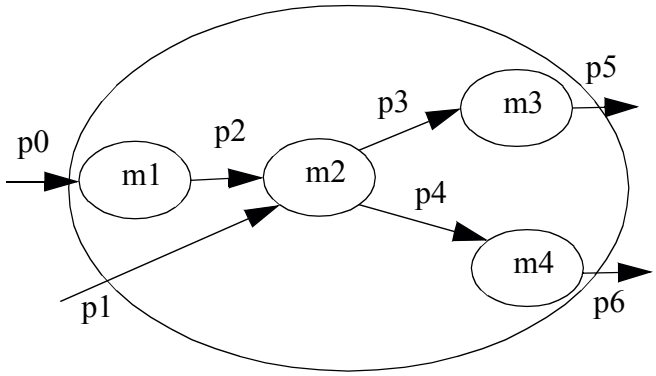


Figure 1. Message flow graph MF

Through closure, MF can be considered as a single message addressed to the framework as a whole, and whose inputs are INPUTS(MF) = $\{p_0, p_1\}$ and whose outputs are OUTPUTS(MF) = $\{p_5,p_6\}$. In general, we can define INPUTS(MF) as the minimal set of parameters that would enable the triggering of all the messages in the flow graph. In this case, these are the input flows that have not been produced by any other message. Similarly, we can take OUTPUT(MS) as the set of all outputs that were produced but not consumed by other messages. Other closure formulas are also possible.

A more useful characterization of message flow graphs takes into account the side effects of the messages being executed, in addition to inputs and outputs. The side effects include state changes for pre-existing objects, and the creation of new objects. Both can be captured using pre-conditions and post-conditions. Pre- and post-conditions follow a somewhat similar pattern to the inputs and outputs, but expressed in logical terms[1]. Short of characterizing those side-effects pre-

1. This is special case of the identity $wp(S_1 \Rightarrow S_2,R) = wp(S_1,wp(S_2,R))$, where wp stands for *weakest pre-condition*, R stands for a desired result (post-condition), and S_1 and S_2 stand for two programs [Dijsktra,1976].

cisely and being able to reason about them analytically, we chose to extend the definition of message output to include the object being acted upon (and possibly modified), and the objects being created. In other words, we view the output of a message as a triplet <receiver, created objects, actual outputs>. Consider the following example:

```
void Dispatcher::initialise() {
     Queue* q;
     q = new Queue;
     initialise(q);
}
void Dispatcher::initialise(Queue* jq) {
     jobQueue = jq;
}
```

With our definition of message output limited to return values, the above sequence (flow) can be represented by the sequence MS = *d.initialise()* $\Rightarrow$ *d.initialise(q)* and INPUT(MS) = {q}, which is not accurate. With our new definition of output, INPUT(MS) = { }.

In addition to the closure on inputs, outputs and side effects, we need to assign a single *name* to each message flow graph. This name, chosen by the framework ‘‘packager’’, is used to refer to the message flow graph when we embed a framework within another.

3.1.2. How to use a framework

As mentioned earlier, in the actual simulation framework, the interfaces Job, Queue and Dispatcher are, in fact, abstract classes. As such, some of their methods are already fully implemented and some are pure virtual. In order to use the framework, we need to subclass these classes with the *option* of redefining existing methods, and the *obligation* to define the pure virtual ones. This kind of information depends on the fact that interfaces are actual classes, and on the semantics of C++. In fact, we consider this as two pieces of information. The first is the interface of the abstract class (without implementation), and the second is an implicit assertion which says that this abstract class provides a partial implementation for this interface. For this reason, we decided not to include information about which methods to redefine and which to implement in the formal description of the framework itself. For the time being, this kind of information is included in a textual attachment.

To create an actual instance of the framework, we need an *instantiation scenario*. The documentation typically shows code fragments of some ‘‘`main()`’’ program with a set of declarations, participant initializations, and *inter-participant connections*. The code fragment below shows an example instantiation scenario taken from [Dumpleton,1994]:

```
...
main()
{
     static OTC_Job job;
     OTC_Dispatcher::initialise();
     OTC_Dispatcher::schedule(&job);
     ...
}
```

Notice the declaration of `job`, and the call to ‘‘`schedule(&job)`’’ which connects job to the job queue of the dispatcher[1].

Typically, the documentation combines instantiation and example usage (in the form of a message sequence) in the same code fragment. We chose to isolate the creation and connection part and package it in a *function* whose parameters are initialization parameters for the various participants. If the framework were to be considered as a C++ class where the participants are data members, this would be a constructor. The difference between a typical constructor and a framework instantiation scenario is the way the various participants are scoped and interconnected. In a regular class, data members are referred to by name, live beyond the call that initialized them, and all have the same life span. In our case, instantiation scenarios must return one of the participants, and that participant must have explicit connections to the others. This is another example of binding between participants that should be avoided, both in specifications and in implementations.

For specification purposes, we choose to specify (‘‘code’’) instantiation scenarios like constructors, i.e. referring to participants by name as if they were data members (see Figure 3). It is relatively easy to go from this kind of specification to actual C++ code that includes local declarations of the members, but the opposite is not true since we couldn’t tell which local variables are participants, and which are not.

3.1.3. When to use the framework

Given all the difficulties discussed in sections 2.3 and 2.5, for the time being we focus on semi-structured textual descriptions such as the ones used for design patterns. Gamma et al. used three descriptors called ***Intent***, ***Motivation***, and ***Applicability*** [Gamma et al., 1995], which convey two pieces of information: 1) what the design pattern achieves in terms of attaining some design goal (e.g. platform independence)-- embodied in *Intent*. and 2) situations in which such a design goal is desirable (*Motivation*, which gives an example, and *Applicability* which gives rules for recognizing such situations). Neither piece of information is derivable, even remotely, from the behavioral specification of a design pattern, and such information has to be entered manually. With application-specific object frameworks, the nature of the participants is indication enough of the opportunity for using a framework, and this becomes more of a search issue.

3.1.4 Extending the framework

The simulation system (SS) framework is used to implement the event-based system (EBS) framework. The two frameworks has several relationships, including:

Addition of new classes participants and/or specialization of existing ones: OTC_Dispatcher and OTC_job facilities are mainly extended by the classes OTC_Event (events) and OTC_EVAgent (agents). An event may be created and sent to an appropriate agent. The delivery

---

1. For readers not familiar with C++, the syntax `OTC_Dispatcher::initialise()`, where `OTC_Dispatcher` is the name of the class, suggests that the method `initialise()` is a *static member* of the class (equivalent to class method in Smalltalk) and that we assume that there is a single dispatcher per program run.

of an event to an agent is made by creating and adding a job to the dispatcher queue with the «queue» method of the event. For an event *e* and an agent *a* the message is «`e->queue(a.id())`». The created job is different from the others of the simulation system. For this purpose, the EBS framework derives a new class from OTC_Job (OTC_EventJob) with some extra facilities to take into account the particularity of event jobs as shown in the following code.

```
class OTC_EventJob : public OTC_Job {
public:
     int target()const {return myTarget;};
     int event() const {return myEvent;};
     void execute(); // Delivers the event to the agent

private:
     int  myTarget; // the ID of the agent.
     OTC_Event* MyEvent; // the event
```

An event can also be sent without creating a delivery job; this is possible by using the «deliver» method of the event. In this case we don't need to use the dispatcher queue.

Extension of message flow graphs: The queued delivery of events to agents uses the simulation framework's basic "run" cycle (select, run, destroy). The created event job is scheduled like any other job type and is selected from the queue. The following code shows the method «queue» of `OTC_Event`:

```
void OTC_Event::queue(int theAgentId)
{    OTC_EventJob* theJob;
     theJob = new OTC_EventJob(theAgentId,this);
     OTC_Dispatcher::schedule(theJob);
}
```

Thus, whereas in the simulation systems (SS) framework we schedule a pre-existing job *j* using the sequence MS_1 *d.schedule(j)* $\Rightarrow$ *q.add(j)*, the event-based system framework enables us to schedule the delivery of an event *e*, i.e., it supports the longer sequence MS_2 *e.queue(id)* $\Rightarrow$ *d.schedule*(j_{id}) $\Rightarrow$ *q.add*(j_{id}). In this case, a sequence of SS is a *suffix* of a sequence of EBS. In the more general case of message *flow graphs* (instead of linear sequences), we can have more complex relationships, based on graph inclusion. For example, the SS framework supports the following message flow graph, MF_1, where the (round) bracket notation means that three piled messages are triggered by *d.dispatch(),* and are executed in the sequence top to bottom:

$$d \cdot run(\,) \Rightarrow d \cdot dispatch(\,) \Rightarrow \begin{pmatrix} i <\text{-}\ q \cdot next(\,) \\ i \cdot execute(\,) \\ i \cdot destroy(\,) \end{pmatrix}$$

The EBS framework, on the other hand, supports a different version of ***execute****,* which, in turn,

triggers a sequence of two messages, yielding the following flow graph (MF_2):

$$d \cdot run(\,) \Rightarrow d \cdot dispatch(\,) \Rightarrow \left(\begin{array}{c} i <\text{-}\ q \cdot next(\,) \\ i \cdot execute(\,) \Rightarrow e \cdot deliver(id) \Rightarrow a \cdot handle(e) \\ i \cdot destroy(\,) \end{array} \right)$$

One of the differences between this extension and the previous one is that the trace of MS_1 was included in the trace of MS_2, while this is not the case for the pair MF_1,MF_2. We define message flow graph extension informally below; we first define message pair specialization:

Message pair specialization:

A message pair x.f(...) ⇒ y.g(...) specializes x'.f'(...) ⇒ y'.g'(...) iff:

- *x' is identical to, or specializes x, and f' is identical to, or conformant to f*
- *y' is identical to, or specializes y, and g' is identical to, or conformant to g*

Message flow graph extension:

A message flow graph MF_1 extends a message flow graph MF_2 iff:

- *For all message pair x.f(...) ⇒ y.g(...) in MF_2, there exists a message pair x'.f'(...) ⇒ y'.g'(...) in MF_1 that specializes it, and*
- *For all message cascade x.f(...) ⇒ $y_1.g_1(...)$ $(OP_i\ y_i.g_i(...))^*$ in MF_2, there exists a corresponding message cascade x'.f'(...) ⇒ $y'_1.g'_1(...)$ $(OP_i\ y'_i.g'_i(...))^*$ in MF_1 such that for all i, x.f(...) ⇒ $y_i.g_i(...)$ specializes x'.f'(...) ⇒ $y'_i.g'_i(...)$*

The operator OP_i stands for explicit sequencing (corresponding to pile notation in the examples shown above) or parallelism.

The two basic relations between framework components support two kinds of relationships between frameworks: *generalization*, and *aggregation*. Generalization is based on the substitutability principle: *A framework F_1 specializes a framework F_2 if wherever F_2 is expected F_1 can fulfill its role.* This means that F_1 has at least the same components as F_2 (or behaviorally conformant ones), and if the message sequences restricted to those common components are equivalent. Formally:

Framework generalization:

A framework F_1 with participants $P_1,...,P_m$ and message flow graphs $MF_1,...,MF_o$ is a specialization of a framework F'_1 with participants $P'_1,...,P'_n$ and message flow graphs $MF'_1,...,MF'_p$ iff:

- *For all $1 \le i \le n$, there exists $1 \le j \le m$ such that P'_j is a specialization of (or identical to) P_i*
- *For all $1 \le r \le p$, there exists $1 \le s \le o$ such that MF'_s extends MF_r.*

Framework aggregation is defined in such a way that references to F_1's participants in F_2 be removed and replaced by a reference to a single participant whose interface is (a subset of) F_1's interface. Let *PART(F)* be the set of participants in a framework, and *PART(MF)* the set of participants. Semi-formally:

Framework aggregation:

> *A framework F_1 with participants $P_1,...,P_m$ and message flow graphs $MF_1,...,MF_o$ is a component of a framework F'_1 with participants $P'_1,...,P'_n$ and message flow graphs $MF'_1,...,MF'_p$ iff:*
>
> - *For all $1 \le i \le n$, there exists $1 \le j \le m$ such that P'_j is identical to P_i*
> - *For each MF'_s of F_2 such that $PART(MF'_s) \cap PART(F_1) \neq \emptyset$, there exists a message flow MF_j of F_1 such $MF'_s = MF_j \oplus MF'_{s,j}$ and $PART(MF'_{s,j}) \cap PART(F_1) = \emptyset$*

In the above definition, $MF'_{s,j}$ is the message flow graph obtained by «gutting out» MF'_s of the message pairs or cascades found in MF_j.

We have just begun to explore the relationships between frameworks found in actual code libraries, and our experience has been that the relationship between any two frameworks to be a complex combination of aggregation and generalization relationships between the frameworks themselves and/or common subcomponents or generalizations. One can imagine a frameworks browser that generates (virtual) common generalizations of frameworks, e.g., for the purposes of navigation.

3.2. Representation model

The previous examples illustrated only some aspects of object frameworks. We introduce in this section the full notation, and discuss those aspects not brought up earlier.

The description of an object framework consists of five descriptive slots, as shown below:

```
Framework <name> {
     Variables:
          Var1 : Type1;
          ...
     Participants:
          Part1 : Interface1 ;
          ...
     Constraints:
          Rel1(Part_i,...,Part_j,Var_m,...,Var_n);
          ...
     Tasks:
          Task1 (Inp1 : IN T1,...,Inp_i: IN T_i,
                    Out1: OUT T'1,...,Out_k: OUT T'_k)
               {
               Part1.f(Inp1) ⇒ Part_j.f(Inp_j) ...;
               ...
               }
          ...
     Instantiations:
          Scenario1 (p1: T1,...,p_i: T_i)
               {
               Part1.initialise(p1);
               ...
               }
```

```
        ...
}
```

The slot *variables* contains variables which are specific to the framework as a whole, but not related to any participant in particular. These could be state or *status* variables, or variables used to bind participants to each other. For a model/view framework where model changes are buffered, a state variable could indicate whether the model and the view are in sync. Another use of variables is illustrated below.

Constraints describe invariant relationships that must hold during the lifetime of the framework. An object framework may go through a transitory phase during which a constraint is not satisfied, but the idea is that if a transaction completes successfully, all constraints should be satisfied in the end. We distinguish between three kinds of constraints, 1) constraints between participants, 2) constraints between a participant and a variable, and 3) constraints between a participant and a constant. Constraints between participants take the form of a relationship between their attributes [Mili, 1996]. In a model/view framework, we can constrain the value of an instance variable of the model to the height of the graphical bar (a view) representing the model in a bar chart (a composite view). Constraints between participants and variables may be used to represent dependencies between framework participants and the outside environment. If the variable represents a sensor, e.g., this would be one way of relating sensed data to the participant responsible for handling it. Constraints between participants and variables may also be used to represent a many-to-many constraint between n participants (e.g. n x(n+1)/2 connections) by a n connections to a common variable.

Constraints and message flow graphs are tightly coupled. Message flow graphs which, transitorily, violate constraints will contain subsequent messages which re-enforces them. Conversely, what would have been a single state-modifying message on a constrained participant, becomes a trigger for an entire message flow graph whose sole purpose is to re-establish the constraint. In the model/view framework, the methods that change the state of model (*m*) variables that have a graphical rendering *have to* call the methods that update the view (*v*) 's corresponding parameter, and its graphical display, as in[1]:

`m.set<an attribue name>(x)` $\Rightarrow$ `v.update(<an attribute name>,x)`

We can go one step further. If we represent a constraint such as:

`m.volume` $\rightarrow$ `v.barHeight`

where $\rightarrow$ means that whenever the left hand side changes, the right hand side has to follow suit, we can imagine a constraint parser that automatically adds a call to `v.setBarHeight(x)` at the end of the method `m.setVolume(x)`. Alternatively (and more easily), the constraint parser can add an after-method to the call to `m.setVolume(x)`.

We believe that given a catalog of such constraints and corresponding transformations, we can account for most of the cases of inter-participant constraints and message connections. This

1. In Smalltalk's MVC, such connections are achieved as follows: 1) views are made *dependents* of models, and 2) whenever a model changes state, it broadcasts a messages to its dependents notifying them of the change, and 3) dependents decide whether to react to the change, depending on the nature of the change. This scheme relieves models from knowing specifically which view method to call in each case.

approach would have several advantages:

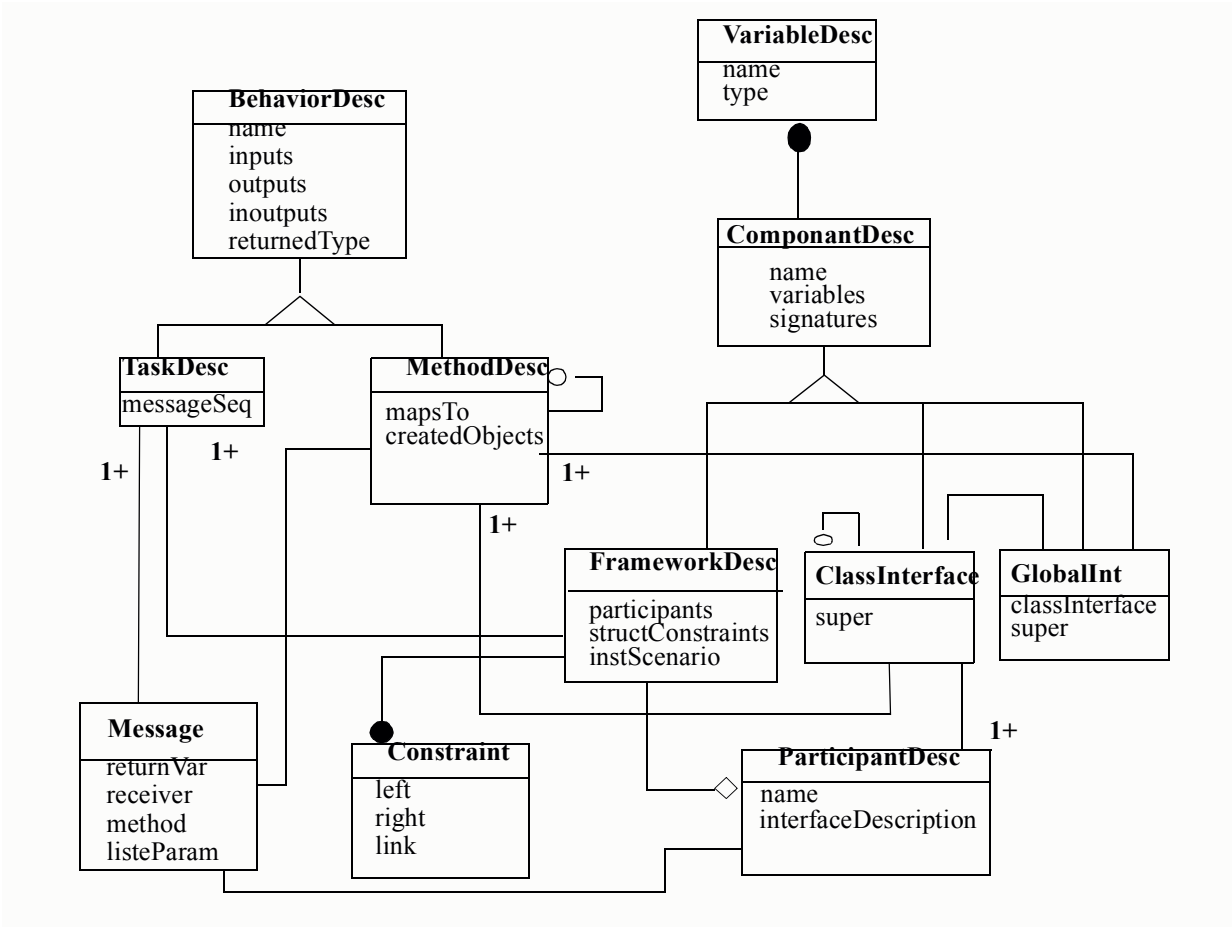
- Simplifying the specification of object frameworks: all we have to do is specify the constraints, and the message sequences/connections will be derived automatically,
- Ensure static connections between the participants without having to create named instance variables that point from one to another,
- Ensuring message connections between the participants implicitly, by avoiding the lexical binding between their methods.

Admittedly, the interconnections within an object framework involve more than propagating state variables. Further, a specification method has to accommodate existing frameworks, and possibly, support the extraction of such specifications from the code; if existing frameworks are not programmed according to this constraints style, it will not be possible to delineate such constraints and their enforcement sequences. Theoretically, however, we can represent *and* implement behavior and behavioral composition using logic and constraint-logic programming languages (see e.g. [Saraswat,1989],[Wilk,1991], [Freeman-Benson,1989]). Further, we have shown in [Mili,1996] that we can support behavioral composition within an imperative language using a combination of constraints and a properly tuned message-sending protocol.

For the time being, we will use the constraint notation for documentation purposes only. Further, we include message sequences in the description of a framework even if all the sequences can be inferred from the specified constraints. In the long run, we advocate a constraint-based inter-object behavioral composition. We are currently developing a set of practical design guidelines based on the results in [Mili,1996], and intend to try them out to re-engineer a number of existing frameworks, including the event-based simulation framework mentioned earlier.

3.4. Prototype framework browser

To date, we have implemented the representation part of our representation of object frameworks and the search engine. Two considerations guided our implementation: 1) integration into a (research) prototype class library tool, and 2) assuring representation uniformity between class descriptions, interface descriptions, and framework descriptions. Figure 2 shows a simplified data model for the representation. For each of the two hierarchies in Figure 2, we have a generic class (BehaviorDesc and ComponantDesc) and two specialized classes, one related to frameworks and the other to participant interfaces. The structure and behavior supported by the generic classes support the closure operations mentioned earlier, and hence the **embedding** of framework descriptions. We discuss below some aspects of the model in more detail. The attribute ‘variables’ of the class ComponentDesc is used to represent both instance variables for regular classes and class interfaces, and framework variables, as explained in section 3.1.2. Signatures accounts for both method signatures (for regular class interfaces) and task signatures (see section 3.1.1 and 3.1.2). The subclass FrameworkDesc has an additional three instance variables corresponding to the participants, structural constraints, and instantiation scenarios, respectively. The representation of actual classes is not shown in Figure 2. Suffice it to say that our library tool supports the representation of various *interfaces* or *views* for the same class[1]. Practically, this means that classes are aggregations of ‘ClassInterface’s. It also means that when we look for potential participants for a chosen framework, we would actually be matching the class interfaces that represent



participants to those that represent views of classes.

Figure 2. implementation model

Figure 3 shows a prototype frameworks editing tool. The top section, with three lists is self-explanatory. The middle section consists of the various participants, along with their interfaces. The currently selected participant, *j*, is of interface **Job**. The lists «Operation signatures» and «Attributes» describe the interface **Job**. The «Tasks» section of the interface lists the signatures of the tasks, and the corresponding message sequences. Messages sequences are first entered in the text area, and then «compiled». The «compiler» prompts the developer for a name and a return variable (hence type), and generates the rest of the signature using the default closure formula (see sections 3.1.1 and 3.1.2). The developer has the option of adding or removing outputs from the signature. The «Instantiation scenarios» section is similar in principle to the tasks section.

We have implemented additional tools for visualizing and editing textual documentation for the

1. This concept is a generalization of C++'s three visibility interfaces, *private*, *public*, and *protected*: each class can have several interfaces which are made available to, possibly different, kinds of server programs

frameworks. Both tools can be invoked through action menu options within the «Frameworks» list.

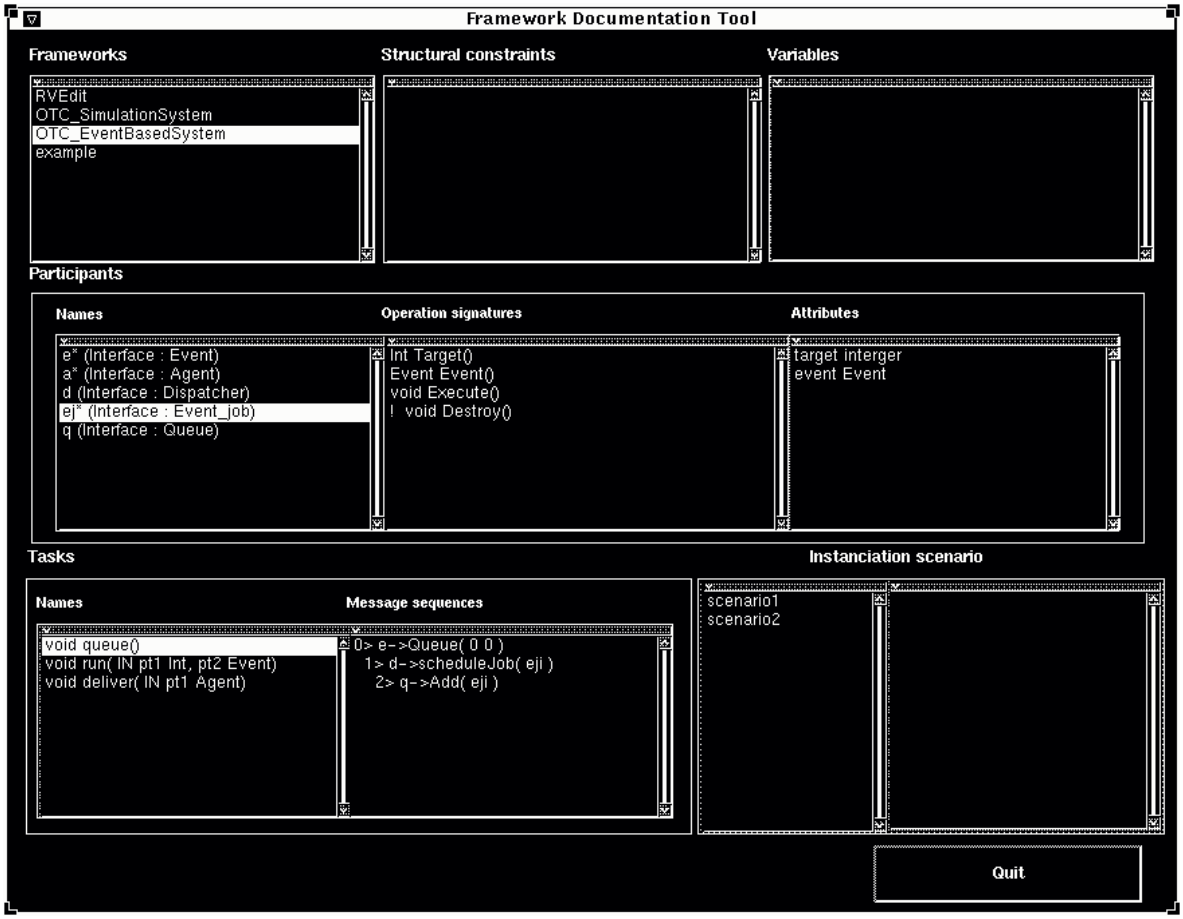


Figure 3. Frameworks browser tool.

4. Framework search and realization

Frameworks are but one kind on reusable components that developers can search for, browse, and integrate into their own applications. We set out to develop a family of search algorithms that can locate object frameworks based on *intrinsic* structural and behavioral information as well as *extrinsic* information such as external textual documentation and a faceted classification. Because we used a representation of frameworks that is similar to that of classes[1], we were able to port those search algorithms that use extrinsic information that we had first developed for class components (see e.g. [Mili et al.,1997]) to work for both kinds of components. Such algorithms

1. The factorization shown in Figure 2 took place after we had implemented class representations.

included string search algorithms and multi-faceted component retrieval algorithms. Experiments with users showed that string-search algorithms performed better than multi-faceted retrieval algorithms [Mili et al., 1997], and were not explored any further in the context of frameworks. We will limit our discussion to search methods on intrinsic information, and more specifically, signature matching algorithms. Query formulation and the actual signature matching are discussed in section 4.1.

Unlike the case of class components, which are reusable as is, frameworks are abstract descriptions of collaborations of classes playing specific roles, and they need to be *realized* or *concretized* using actual domain or library classes. Realizing a framework involves finding or creating, for each participant interface, a class with the proper role in the underlying application (domain) that satisfies the interface. Framework realization also involves interface matching, and we will discuss potential problems due to cyclic type references and ways to address them. Framework realization is discussed in section 4.2.

4.1 Formulating and matching interface queries

Referring to the two-stage search discussed in section 2.5, the first stage consists of searching for a framework that satisfies a number of criteria. We implemented a framework matching algorithm based on *signature matching*. Broadly speaking, a developer specifies a class-like interface for which he wishes an «implementation», and the system looks for reusable components that support the interface. Those components could be either single classes or frameworks whose message sequences and instantiation scenarios have been abstracted or «closed» into signatures. We will first describe query specification, and then the actual matching.

Figure 4. Interface matching query tool.

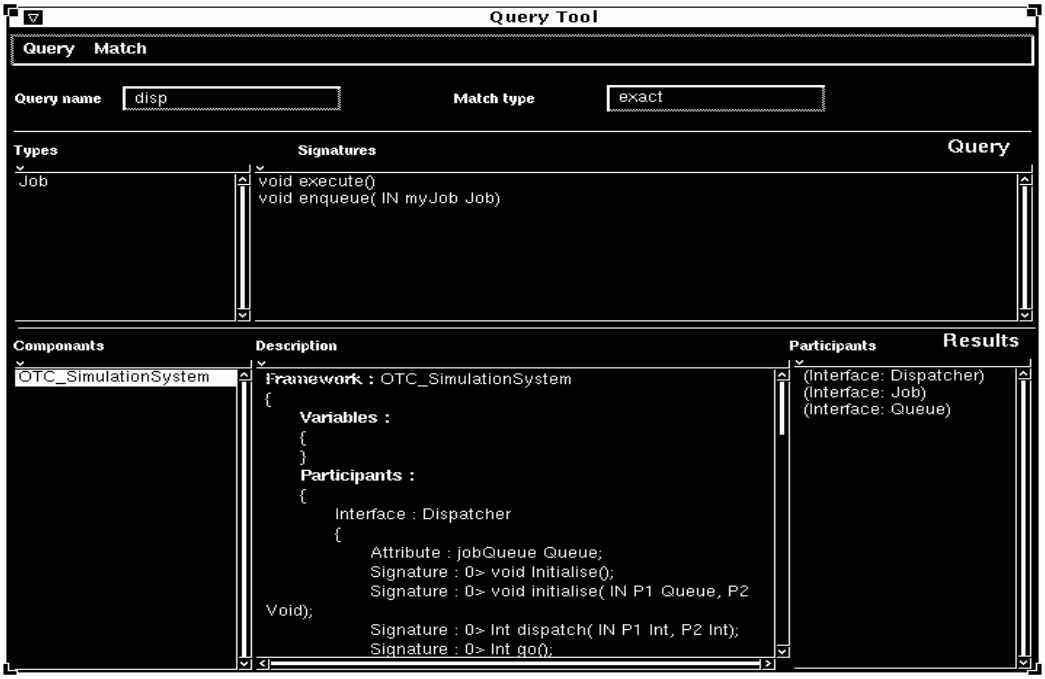
Figure 4 shows the query interface tool. A developer specifies the query in the upper part of the window, and the results are shown in the bottom part. A query consists of a set of method signatures, along with those types that are referenced in operation signatures, and *that are not in the library*. For those types, the developer needs to specify a list of equivalent types/interfaces from the library so that matching can take place. In the above example, if the developer has *asserted* that Job was «equivalent» to any of OTCJob, OTCEventDelivery, or OTCProcess, then we would be looking for any component that supports a method with signature

```
void execute()
```

and any *one of* the following signatures:

```
void enqueue (IN myJob OTCJob), or
void enqueue (IN myJob OTCEventDelivery), or
void enqueue (IN myJob OTCProcess)
```

Deciding on the equivalence of Job to OTCJob, OTCEventDelivery, and OTCProcess, may itself be the result of an interface matching operation: upon realizing that Job is not defined, a developer may choose to spawn another search window, and specify the interface of the new type using method signatures. This potentially recursive process has to have a termination: either the devel-



oper specifies an interface in terms of known types, and the search tool is able to match it to existing components, or s/he asserts the equivalence of types explicitly.

The actual interface matching ignores method names, and uses type matching. We support two variations of the matching algorithm, an *exact match*, which takes into account parameter type positions, and *inexact match*, which does not take into account positions. Matching can return either a single class interface or a framework interface. The «Description» subpane shows a textual representation of the interface. For the case of a framework, the textual representation is generated from a template using the structural representation of the framework. For the case of a class, the textual representation consists simply of the C++ header file. The «Participants» subpane lists the set of participants-- the case of frameworks-- or data members/instance variables, for the case of classes. The example of figure 4 shows a framework interface.

4.2 Framework realization

Recall that a framework is a *design idea* or *an execution pattern* and not an executable piece of code. Hence, once we have found a framework that satisfies some external behavior, we have to find a *realization* or *specific implementation* of the framework within the context of the application at hand. Formally, we have to find implementations for the participant of the framework. A *complete* (*partial*) realization is one where we find an implementation for *each* (*some* of the) participant in the framework; the implementation of a participant may be found in the library of components, or may have to be constructed. In general, the participants that play an application independent role are provided in the library, while the ones that depend on the application at hand are to be constructed or otherwise provided by the framework user. For example, in Smalltalk's Model-View-Controller framework, only the model classes have to be provided by the framework

user (developer); the Smalltalk class library contains classes that implement most of the common view and controller behaviors (modulo few parameterizations).

Symbolically, let $\{I_1, I_2,...,I_n\}$ be the set of participant interfaces to be matched and $Imp(I_i) = \{C_{i,1},...,C_{i,ki}\}$ the set of classes that match the interface I_i. A realization of the framework consists of a tuple $<C_1,...,C_n>$ where $C_i \in Imp(I_i)$. Note that we could have $I_i = I_j$ for some $i \neq j$ because the framework can have more than one participant with the same interface. Further, for a given pair i,j such that $I_i = I_j$, we could have $C_i \neq C_j$ if we choose different implementations for two participants that have the same interface.

In existing frameworks, the distinction between participant specifications (interfaces) and actual implementations is not very clear as a *class* would represent both. Further, language properties such as typing and early versus late binding, and programming style, may further confuse matters, making interfaces and implementations virtually inseparable. For example, in a language such as C++, participant interfaces are embodied in *abstract classes* from which actual participants have to inherit. To complicate matters further, the *abstract classes* themselves may be partially implemented, and only a handful of methods may have to be defined in the derived subclasses. Further, the participants of a framework refer to each other, as in the case where the method of one participant uses another one as a parameter. In typed languages such as C++, where subclassing is used as the subtyping mechanism, we are forced to represent even the application-dependent classes by actual C++ *abstract classes* from which the framework users have to derive the actual classes[1].

Within the context of our prototype, finding the implementation of a particular framework participant involves performing signature matching between the interface of the participant and interfaces of *concrete library components*. A *concrete library component* is a component that has a source code equivalent. Typically, most of the components in the library are concrete and are obtained by parsing source code files and loading the resulting structures in the tool set (see e.g. [Mili et al., 1997]); non-concrete components may appear as participant interfaces in framework descriptions, or framework interfaces themselves. In practice, referring back to the object model of Figure 2, for each C++ class read in the input, we create an instance of the class **ClassInterface** that represents the locally defined (or redefined) members of the class, and that points, among other things, to the actual source code. The actual interface/type of the class includes not only locally (re-)defined members, but also inherited ones, and is only computed when needed during matching[2].

For the case of C++ frameworks, the interface of a framework participant-- a non-concrete library component-- may match the interface of a C++ *abstract class* C_a-- a *concrete library component* according to the above definition. If we choose such a class in the implementation of the framework, we still wouldn't get an executable framework realization. However, if during framework instantiation, we use an instance of a *concrete subclass* (in the programming language sense) of C_a, we will be fine. Further, we may be forced to use only subclasses of C_a for type compatibility

1. In Smalltalk's MVC, the class library contains an «abstract class» called **Model** although developers need not subclass it for their application-specific classes.
2. One of the consequences of this on-the-fly computation of interfaces is that we don't have the notion of type identity («equality», ==); instead, we have type equivalence (=).

reasons. For instance, to obtain the formal/abstract description of the framework OTC_Simulation_System discussed earlier (see sc. 3.1), we abstracted code level dependencies between the participants by replacing classes by the types they implement. However, we cannot obtain a realization of the framework by taking *any* combination of implementations of each of the participants; this is one of the problems that give rise to the *factory method* or *factory class* pattern (see e.g. [Gamma et al., 1995]). For example, the class Dispatcher (which «implements the interface» Dispatcher) refers to the *abstract* class Job in its *source code*. Thus, if we use the class Dispatcher in the simulation system framework, we have to-- from a programming language point of view-- use a subclass of the abstract class **Job**. Symbolically, this means that if $\{I_1, I_2,...,I_n\}$ is the set of participant interfaces to be matched and $\mathrm{Imp}(I_i)$ is the set of classes that match the interface I_i, the set of realizations $<C_1,...,C_n>$ of the framework, where $C_i \in \mathrm{Imp}(I_i)$, is *not* the cartesian product $\mathrm{Imp}(I_1) \times ... \times \mathrm{Imp}(I_n)$, but rather a *proper subset* thereof.

What all of this means is that, notwithstanding the lack of behavioral considerations in interface matching, code-level dependencies further weaken the effectiveness of interface matching as *sufficient* conditions for the realizability of a framework. Of all the classes whose interfaces match that of a participant, we have to

- Exclude abstract classes-- because they are not fully implemented-- and
- Make sure that the concrete classes that we use are subclasses of the abstract classes that are explicitly referred to in the source code of the other participant implementations.

This last condition creates a circular dependency between the participants, but that is a reflection of the kind of dependencies that exist within existing framework implementations. This circularity of reference may manifest itself at the interface level. Consider a framework with two participants with interfaces $I_1 = \{f(int,I_2)\}$ and $I_2 = \{g(int,I_1)\}$, and the classes $C_1 = \{f(int,C_2)\}$ and $C_2 = \{g(int,C_1)\}$. We can assert that C_1 matches I_1 only if we can assert that C_2 matches I_2 and vice-versa. This is a constraint satisfaction problem whose solutions are tuples of mutually coherent participant implementations. Our current implementation of the prototype does not perform this global realization process, and the developer has to choose a realization one participant at a time. In case of a non-circular chain of dependencies, the developer has to find an implementation for the first element of the chain, and from that point on, the system is able to suggest the others. In the example shown in Figure 5, we were only able to assert that the class OTCDispatcherGI matches the interface Dispatcher because we had already established that OTCJobQueue (2nd line of the left side 'signatures' list) matched the interface Queue, and that OTCJob (10th line of the same list) matched Job; if we hadn't, we wouldn't even have been able to suggest/suspect that OTCDispatcherGI is a candidate for the interface Dispatcher.

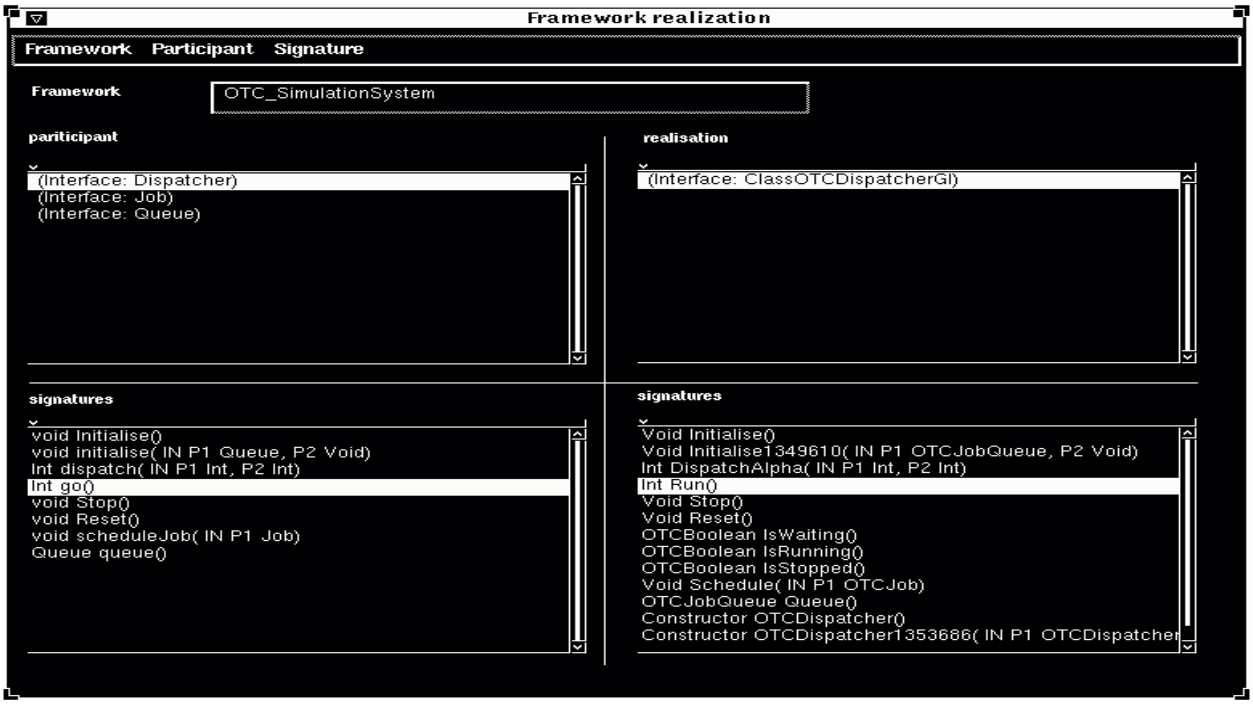


Figure 5. Framework realization interface.

We find this dependency on the order in which the participants are «realized» unacceptable, and we introduced the concept of *conditional matching* whereby an implementation is presented as satisfying an interface *if* some <interface, implementation> pairs are shown to match. For example, consider the interface I_1 = {f(int,I_2)} and the class C_1 = {f(int,C_2)}, we can say that I_1 matches C_1 if (and only if, in this case) C_2 can be shown to match I_2, or

$$\texttt{Match}(I_1, C_1) \Leftarrow \texttt{Match}(I_2, C_2)$$

For each participant interface *I* and library class *C*, the expression 'Match(*I*,*C*)' can have 4 possible values:

- *unknown*, in case the pair has not been evaluated,
- (*provably/proven*) *false*, in case it has been evaluated and was found to be false, independently of everything else,
- *conditionally true*, in case it was evaluated, and was found to be true provided that Match(*I'*,*C'*), for some pairs <*I'*,*C'*> where *I'* was referenced in *I* and *C'* was referenced in C, was either unknown or found to be conditionally true, and
- *true*, if it was evaluated and was found to be true.

When the value of an expression Match(*I*,*C*) is changed, from 'unknown' to one of the other val-

ues, or from 'conditionally true' to either 'true' or 'false', that change is reflected on the conditionally true expressions Match(*I'*,*C'*) that depend on Match(*I*,*C*). In essence, we have a truth maintenance system/network that is updated through developer actions. Under this new version, the matcher still does not attempt to find a global solution on its own, but at least the developer will have, from the start, all the information s/he needs to find such a solution, one participant at a time, and will be guided by the dependencies through the process.

Note that a given class C_i can match an interface I_i in several ways, and have a different justification/precondition for each one. Consider the interface I_1 = {f(int,I_2), g(int, I_3)} and the class C_1 = {h(int,C_2)}. C_1 can match I_1 in one of two ways:

$$\texttt{Match}(I_1, C_1) \Leftarrow \texttt{Match}(I_2, C_2), \texttt{or}$$
$$\texttt{Match}(I_1, C_1) \Leftarrow \texttt{Match}(I_3, C_2)$$

which correspond to mapping function f(.,.) to h(.,.) versus mapping g(.,.) to h(.,.). When we present a solution to the developer, we have to specify the exact mapping between types and implementations. In general, a single component realization may itself be seen as a set of pairs {<I_1,C_1>,...,<I_n,C_n>} where Match(*I*,*C*) evaluates to true for some pairs, and to conditionally true or unknown for the others. We are currently experimenting with variants of the rudimentary interface shown in Figure 5 to integrate the new information (specific type mappings for a given <*I*,*C*> pair, and their status), and to offer the required functionalities to incrementally build and visualize the framework realization.

4.3 Discussion

When we set out to develop a representation of frameworks, applicability to existing frameworks was a major concern. Another overriding concern was a non-commitment to an interaction/coordination mechanism between framework participants, hence the idea of message sequences to represent inter-object behavior. This representation is well-suited for describing and reasoning about frameworks; it breaks down, however, when we deal with the actual realizations because of hidden/abstracted code-level dependencies between participant realizations.

We have long advocated (see e.g. [Mili et al., 1990], [Mili, 1996]), along with others (see e.g. [Borning,1986], [Freeman-Benson,1989], [Wilk,1991]), a declarative inter-object behavioral composition mechanism, as a way of building independently reusable objects, and of providing greater possibilities for behavioral composition [Mili,1996]. Roughly speaking, objects respond to messages by executing methods and by notifying other interested parties (other objects or 'constraints') of any state changes that may have resulted. This notification will in turn trigger other objects to execute other methods, and so forth. What makes this mechanism effective is the fact that the notifier need not know explicitly the identity of the objects to be notified (their ids or even their classes), nor the specific actions (methods) to take in response to those notifications: this information resides in an external 'table'[1] that can be updated programmatically, instead of being hard-wired in the source code of the participants. This paradigm serves as the basis for many user interface frameworks (see e.g. [Barth,1986], Smalltalk's MVC), but rarely followed to the letter in industrial strength frameworks, in part for efficiency reasons. Our structural model of frameworks provides for the representation of such dependencies (*constraints* and, to some extent, *variables*,

1. much like a dispatch or an interrupt table

see section 3.2) and we can imagine either a «framework compiler» that would translate such constraints into explicit calls between framework participants, or a «run-time dispatcher» that would assure the coordination during execution.

Other behavioral composition paradigms could be considered, but they too, would involve a different programming style. One such paradigm consists of coding framework participants as parameterized/generic types *à la* Ada's generic packages or C++'s templates (see e.g. [Tracz,1993]). For example, using C++'s templates, we could define the class Dispatcher as follows:

```
template<class Job> class Dispatcher {
     ...
     void schedule(){
          if (_currentJob->getPriority() > ...)
               ...
     }
     ...
private:
     Job* _currentJob;
     ...
}
```

In this case, we can use any actual/concrete class in lieu of the type parameter Job *as long as it supports the methods invoked on it from* Dispatcher*'s methods* (e.g. the method 'getPriority()' in the above example). This solution does not limit us to subclasses of some predefined abstract class, as was the case with the OSE simulation framework discussed earlier. However, the specific realization of Job has to provide methods with the appropriate signatures *and names*. Artifacts such as *pluggable adaptors* (e.g. Smalltalk's MVC library) or *software connectors* (see e.g. [Yellin &Strom, 1994]) have been proposed as a way of bridging nomenclature and other kinds of mismatches. They may be used to support any composition technology, but are most valuable for the ones that have to rely on source-level dependencies.

Notwithstanding the algorithmic difficulties inherent in signature matching, we are finding out, through simple tests with the OSE library, that signature matching is weak in general when dealing with utility-like classes, as opposed to domain-specific ones, and even more so for a loosely typed language like C++ where the basic types 'int', 'char', and 'void *' (generic pointer) are heavily overloaded for genericity purposes (see e.g. [Mili, Marcotte & Kabbaj,1994]). The example of Figure 5 shows several functions with no input parameters and no return types, or a boolean return type[1]. Obviously, some behavioral descriptions of some sort must be incorporated in the matching to strengthen it. The simplest (and least analytical) way uses method *names* as well as their type signatures, for the purposes of matching, based on the assumptions that names reflect semantics, with known disadvantages (see e.g. [Mili et al., 1995]). For the time being, we are developing an algorithm that uses names to *rank* rather than *filter* the candidates that match based on types alone. The name matching relies on some lexical processing to increase the possibility of matching (e.g. reducing method names to word stems, see e.g. [Mili et al.,1997]).

1. OTCBoolean, which is essentially an int.

A second alternative uses the signatures of the methods called within the methods of a particular class, as an additional matching criterion. For instance, a framework's message flow graphs provide that information for the participant interfaces: the set of messages immediately (as opposed to transitively) triggered by a method of the participant. As for the class components that are present in the library, the parser that we developed extracts call graphs, and the information is readily available [Mili et al., 1997]. Implementing this alternative is a bit more involved, in part because of the circular dependencies, and has not be evaluated yet. Because the above two alternatives rely on the basic type-based signature matching, we will be offering developers with all three variants (vanilla flavor type-based signature matching, type-based with names, and type-based with signature matching of *called* methods), with the last two helping to filter out spurious matches or breaking ties. Later versions may accommodate some variations in the signatures such as optional parameters, type conformance instead of type identity, etc. (see e.g. [Zaremski & Wing, 1993], [Yellin & Strom,1994]); only experimentation will tell if the heavier machinery is worth the trouble.

5. Framework packaging

One of the goals of our representation model was «scalability», or «embeddability», whereby we try to package frameworks in such a way that they can be participants of other frameworks. In this section, we are concerned with the packaging of the source code itself of a framework realization into a single unit. We first look at the motivations behind the packaging, and then the problems it raises and the way we addressed them in the context of our current implementation. It should come as no surprise that we will package framework realizations as classes, and we will discuss the advantages of such a packaging by referring directly to class properties.

Packaging framework realizations as classes is a good way of physically packaging framework-related source code. For instance, a framework may involve the creation and management of data entities other (framework variables) than the participants themselves, which need to «live» and be accessible during the «operation» or «lifetime» of the framework. We need a construct to group these variables, and a class offers the right combination of visibility scoping and lifetime scoping. Further, such classes may be used to encapsulate framework-specific processing such as message sequences and instantiation scenarios.

Packaging frameworks as classes is also good for reuse purposes, as it abstracts away the implementation details of the framework, making it easier to program with the framework, and shielding the resulting programs from any changes in the underlying structure of the framework. Consider the case of the event-based simulation framework shown in Figure 3. Once an instance of the framework is realized, the developer need not be aware of the existence of a dispatcher, of a queue, or the creation of jobs to deliver events, and the assignment of those jobs to agents, as the list of tasks and the corresponding parameters shows (see Figure 3). By programming directly into the framework's interface, we shield our programs from iterative refinements of the framework

which may lead to redistributions of functionalities among the participants, splitting the functionality of a participant into two new participants, offering a wider range of behaviors, etc.

Naturally, one might argue that by wrappering all the functionality into a monolithic class, we are defeating the benefits of building an event based system as a collaboration of several classes, which are, 1) the independent reuse of those classes, and 2) the flexibility obtained by breaking down the functionality, providing a wider range of behaviors. We are *not* defeating those benefits because we envision framework packaging as *a delivery mechanism* and not as a *construction mechanism*: we still develop and maintain (and reuse) participants separately, and *combine them at will for framework realization*. However, if we were to do any maintenance on the framework's code, it has to be done at the level of the individual participants, and the framework realization's class wrapper has to be regenerated.

Finally, by packaging frameworks as classes we isolate the inter-object behavioral composition mechanism, with benefits for both the framework builders and its users. For example, if the message triggering inherent in message sequences/flow graphs is implemented by direct function call, we can code tasks to consist of a call to the first message on the sequence. For example, the task 'schedule' for the simulation system which is defined as follows:

```
void schedule (IN Job j)
{
     0> d->scheduleJob(j)
          1> q->add(j)
}
```

where d stands for the dispatcher and q for the queue, says that the invocation of the message 'scheduleJob()' on the dispatcher «triggers» the invocation of the message 'add' on the queue. Translated into C++ and using function calls, this becomes[1]:

```
void OTCSimulationSystem::schedule (Job j)
{
     d->scheduleJob(j);
}
```

If the dispatcher (d) didn't know explicitly about the queue, and had some processing to do, independently of the actual queueing of the job, the generated code could be:

```
void OTCSimulationSystem::schedule (Job j)
{
     d->scheduleJob(j);
     q->add(j);
}
```

This would be the case, for example, if we used a notification-based mechanism for coordinating participants. If we knew that, i) the dispatcher had to notify its dependents in case new jobs are passed on to it, ii) that the queue was one of those dependents, and iii) that the queue had to add new jobs to itself upon hearing about them, then the above generated code would accomplish that, and would do so *efficiently*.

1. or we should say, «reverts back to», since the message sequences/flow graphs are abstractions of C++ calls.

Note that Smalltalk supports a notification-based mechanism for object coordination (called *dependency mechanism*), and the environment maintains a list of dependents for each object, that are notified «anonymously» when the objects wishes to inform the «world» about changes it underwent. While this mechanism is very flexible, it can be fairly inefficient because *all the dependents* of a particular object are notified whether the changes taking place in that object concern them or not. Basically, what we are able to accomplish by generating code for message sequences/flow graphs is ***provide efficient implementations for flexible behavioral compositions mechanisms***: we could still develop the framework participants (construction time) based on a notification-based communication model, thus removing any code-level dependencies (cross references) between them. When we use them (realization and packaging time), however, we use an efficient substitute for notification-based coordination! This is perhaps the most important advantage of framework realization packaging.

We have developed a first version of a C++ framework packager which maps «abstract» framework descriptions and specific participant realizations into actual C++ code. Figure 6-a shows excerpts of the generated class definition for the framework simulation system. We first summarize the most important transformations from framework description to source code, and then comment on specific constructs. Roughly speaking, the generation rules for the class declaration are as follows:

- Framework → class, where <class name> = ‘‘FP’’ .<framework name>
- Framework participants → data members, where
  - <member name> = <participant name> and <member type>, and
  - <member type> = pointer to the class matching <participant interface>

  When the participant is a collection of objects (e.g. j*, where * is the Kleene star), the member type is a container class capable of holding the type of the participant.
- Tasks → function members where:
  - <function name> = <task name>, and
  - <function signature> = <task signature> where we substitute interfaces by realization classes, OUT (parameter passing mode) by reference passing (&)
- Instantiation scenarios → constructors, where
  - <constructor signature> = <instantiation scenario signature> where we substitute interfaces by realization classes

  In case we have several instantiation scenarios with the same signature (the case of the example of Figure 6), we define a parameterless constructor, but several initialization functions with automatically generated names.

Note that the container class used for «multi-valued» participants depends on the contents of the library. Our packager has options for defining default container classes-- in this case the template class OTC_Collection<T>. Other language-specific default options should be specified, such as the default mapping of built-in types/interfaces[1], ways of handling OUT parameters (as ‘&’ versus as pointers), etc.

1. We use the «abstract» type ‘Integer’ in framework descriptions. It can be mapped to a C++ int, long, unsigned, etc.

```
class FPOTC_SimulationSystem {
public:

     // constructors
     FPOTC_SimulationSystem() ;

     // initialization methods

     virtual void initialize_scenario1();
     virtual void initialize_scenario2();

     // access methods

     OTC_Dispatcher* getDispatcher();
     OTC_JobQueue* getQueue();
     ...

     // tasks

     virtual int runSimulation();
     virtual void schedule(OTC_Job* theJob);

private:
     // data members
     OTC_Dispatcher* dispatcher;
     OTC_JobQueue* queue;
     OTC_Collection<OTC_Job>* jobs;
}
```

Figure 6-a. Excerpts from the framework package class declaration.

The actual code generation involves additional rules, which may be divided into general rules, policy rules, and language-specific rules. General rules include the substitution of interface names by actual class names, and of message names by the corresponding realization class method names. Policy rules have to do with things such as the underlying message sequencing mechanism, e.g. deciding whether each message invokes the immediately subsequent one(s), as is done here, versus using a notification-based mechanism. Language specific rules are self-explanatory, and deal with message sending syntax, iteration constructs, and the like.

```
FPOTC_SimulationSystem::FPOTC_SimulationSystem()
     :jobs(new OTC_Collection<OTC_Job>)
     {}

void FPOTC_SimulationSystem::initialize_scenario2()
{
     dispatcher = new OTC_Dispatcher;
     queue = new OTC_JobQueue;
     dispatcher->Initialise(queue);
}
```

```
...

int FPOTC_SimulationSystem::runSimulation()
{
     return dispatcher->run();
}

void FPOTC_SimulationSystem::schedule(OTC_Job* theJob)
{
     dispatcher->schedule(theJob);
}
```

Figure 6-b. Some method definitions for the framework package class.

The source code generated for the framework tasks and instantiation scenarios is inevitably incomplete because our notation for message flow graphs does not include all of the control information (alternation, loops), and should not be cluttered with low-level operations. Users of a framework package should derive from the generated class, and redefine locally the methods whose generated code is incomplete, which explains why the function methods that correspond to tasks and instantiation scenarios are all declared virtual (Figure 6-a).

At the time of this writing, we have just completed our first version of the framework packager, and we don't have yet practical experience to evaluate the usefulness of the generated method bodies. The important issue, from a maintenance and a reuse point of view, is to figure out whether the manually written code in developers' derived classes is invalidated by a regeneration of the base class. Some of the work we are currently pursuing deals with identifying the simple cases of changes in the framework structure that would have no effect on the derived code, and/or, localizing, as narrowly as possible, the scope of those effects.

6. Related work

There has been a lot of interest recently in object frameworks of various forms, including some of the work mentioned in section 2. Work on the specification, development, and use of object frameworks can benefit from the following areas:

Representation of behavioral compositions: Interest in describing functional or *behavioral* compositions predates the object-oriented paradigm, and some of that work has carried over to the object-oriented domain. There is a wide-spectrum of composition paradigms, depending on the level of abstraction of the language and its granularity. There is a range of *module interconnection languages* [Hall & Weedon, 1993] from the fairly formal LIL (see e.g. [Goguen,1986]) and other descendants of OBJ, which focused on algebraic specification of parameterized data types, to the more practical languages or language extensions which are concerned mostly with verifying that a set of collaborating modules satisfy each other's expectations in terms of supported interfaces (see e.g. ANNA [Tracz,1993], *contracts* [Helm et al., 1990]). Researchers have shown that this requirement is not only needlessly strict (e.g. [Yellin et al., 1994]), but also insufficient, as a lot of the assumptions about the behavior of peer modules are not explicited in their interfaces (see e.g.

[Garlan et al., 1995]). On the more abstract level, we have a range of logic-based composition techniques embodied in constraint and logic programming languages (see e.g. [Saraswat,1990], [Wilk,1991]). These languages have the advantage of supporting composition while obviating the need for cross-referencing (so called *lexical binding*) between components. However, they are too far removed from procedural (practical) languages and are of limited use in practice. Work on reactive and discrete-event systems benefits from a long tradition and a number of formal results on both the validation and verification aspects of compositions [Mili et al., 1995], but is too cumbersome for more traditional applications. Our own approach to *constraints* combines the declarative and (composability) aspects of declarative-style compositions, with an imperative object-oriented programming style (see [Mili et al., 1990], [Mili, 1997]).

Search and realization issues: Software reuse research has traditionally focused on packaging and search issues for reusable components [Mili et al., 1995]. The sharing of reusable component libraries over the internet[1] will bring search issues back to the forefront as developers now have the option of choosing between components from numerous, heavily overlapping, component libraries. Within the context of the two-phase search (see section 2.5), we see two kinds of issues, 1) issues of *closure* as we have to match individual behavioral specifications to compositions of such specifications, 2) the issue of finding potential participants for a given framework. The first problem is a notoriously difficult one (see e.g. [Lam & Shankar,1992]), and we may have to settle for the verification of necessary but insufficient conditions, or use heuristic techniques which rely on structured documentation. The second set of issues is an instance of the more general problem of matching class specifications to class implementations, which has been studied under various forms both theoretically (see e.g. [Guttag et al., 1985], [Goguen,1986], [Chen et al., 1993], [Mili et al., 1994a]) and practically (see e.g. [Zaremski & Wing, 1993]). At issue are both the decidability of interface matching itself, as well as the usefulness of matches. Typically, the more information used in the matching process, the more significant the matching. However, the additional information often comes at a great expense, or relies on unrealistic assumptions. We have been careful to limit our investigation to constructs that are readily available in existing frameworks code, or to ones that can be easily extracted.

Packaging issues: the problem of packaging a set of interacting objects/modules into a module arises quite often in OO work, be it at the modeling/analysis level (e.g. [Mili, 1990], [Lieberherr et al., 1991], [De Champeaux et al.,1993]), or at the code level with things such module interconnection languages ([Hall & Weedon, 1993]). Some of the issues include deciding on what to expose to the outside world, and what to hide, as well as providing the same kind of flexibility through the encapsulation interface. Interestingly, much of the work on packaging frameworks seems to come from the practical arena with the revival of code generators in general, and user interface generators in particular. While much of the work on user interface frameworks during the 80's focussed on better delineation of user interface functionalities and separate packaging of those functionalities (e.g. the MVC framework [Reenskaug,1995], Interviews [Linton et al., 1989]), the newer generation of development tools and environments (see e.g. ParcPlace/ Digitalk's VisualWorks and Parts product lines, IBM's VisualAge/Smalltalk environment) focus on providing developers with the means to specify the «external» behavior and appearance of

1. The library from which the two example frameworks was retrieved via a rudimentary web keyword search.

interfaces, but relieve them from the implementation details of the frameworks by generating automatically the glue code that ties the MVC pieces together[1]. In essence, this trend is moving frameworks from a paradigm for the *user* of reusable components, to a paradigm for the *developer* of such components. Issues of maitainability of frameworks and the inter-operability of frameworks that share participants need to be addressed, and we have barely scratched the surface.

7. Discussion

In this paper, we described ongoing research at the University of Quebec to develop a representation for object frameworks that supports a number of often conflicting goals, not the least of which is our desire for solid theoretical underpinnings and immediate applicability. Our theoretical work progress hand in hand with prototyping efforts which aim at making our ideas workable. Our research strategy has been to cover the entire lifecycle of development with frameworks-- albeit not at great depth-- with the purposes of identifying the major issues that need to be addressed so that we can provide effective and robust support for programming with object frameworks.

From a theoretical standpoint, there are a number of outstanding issues. First, we are exploring the range of useful relationships that may exist between object frameworks and that support extensibility and scalability. We have seen in section 3.1.4 two examples of such relationships. Along with exploring the various kinds of relationships, we will explore design or style guidelines that would make such frameworks more “separable”. Second, there remain a number of outstanding issues with the packaging of message flow graphs as single messages through *interface closure.* For the time being, we have used default «conservative» closure rules. Notwithstanding the fact that such a closure may carry extraneous parameters, it ignores the inter-play of pre- and post-conditions between the messages in the flow graph, and may lead to inconsistent or indeterminate parameter values. Further, our closure algorithm abstracts out some aspects of object synchronization, and creates atomic boundaries around essentially non-atomic operations. This means that the *interface* closure of a task may not always be a faithful abstraction of what happens within the task; additional *behavioral/dynamic* information may need to be included. This problem is not specific to frameworks; it arises with reusable software of all sorts [Kiczales & Lamping,1992], [Garlan et al., 1995]. Finally, we are also experimenting with the generation of message sequences from the specification of structural constraints (see section 3.2). Key to this effort is the construction of a catalog of such constraints.

From a practical standpoint, we are developing tools for automating the extraction of framework specifications. Other aspects of our work required that we build a C++ parser that uses static type analysis to extract static cross-references between methods belonging to various classes. We are augmenting this parser to extract framework specific message flow graphs by: 1) limiting the call graphs to those that are initiated by methods that are part of the required interfaces of framework classes, and 2) by performing some measure of data flow analysis to trace actual framework participants (as opposed to their types) through a call graph; tracing a variable through casting and polymorphic assignment is proving to be quite a challenge. We are also refining the tools already

1. Which makes it even harder to teach good OO design practices with such tools and environments.

developed. First on our agenda is the interface matching tool in general, and the framework realization tool in particular. We are currently working towards a realistic data set so that our methods and tools can be validated experimentally.

References

[Allen & Garlan, 1994] Robert Allen and David Garlan, ‘‘Formalizing architectural connection,’’ *Proceedings of ICSE’16,* May 16-21, Sorrento - Italy, pp. 71-80.

[Barth,1986] Paul Barth, ‘‘An Object-Oriented Approach to Graphical Interfaces,’’ *ACM Transactions on Graphics*, Vol. 5, No 2, April 1986, pp. 142-172.

[Borning, 1986] Alan Borning and Robert Duisberg, ‘‘Constraint-Based Tools for Building User-Interfaces,’’ *ACM Transactions on Graphics*, Vol. 5, No. 4, October 1986, pp. 345-374.

[Chen et al., 1993] Patrick S. Chen, Rolh Hennicker, and Matthias Jarke, ‘‘On The Retrieval of Reusable Components,’’ in *Advances in Software Reuse,* selected Papers from the Second International Workshop on Software Reusability, Lucca, Italy, March 24-26, 1993, IEEE Computer Society Press, pp. 99-108.

[Darden & Rada, 1988] Lindley Darden and Roy Rada, “Hypothesis Formation Using Part-Whole Interrelations,” *Analogical Reasoning: Perspectives in Philosophy and Artificial Intelligence,* Eds. Helman, Reidel & Dordrecht, Netherlands, 1988.

[De Champeaux, 1993] Dennis De Champeaux, Doug Lea, and Penelope Faure, *Object-Oriented Software Development*, Addison-Wesley, 1993. See in particular chapter on Ensembles (pp. 133-143).

[Deutsch,1989] Larry P. Deutsch, ‘‘Design Reuse and Frameworks In The Smalltalk-80 Programming System,’’ in *Software Reusability*, vol II, Eds Ted J. Biggerstaff and Alan J. Perlis, ACM Press, 1989.

[Dumpleton,1994] Graham Dumpleton, *OSE - C++ Library User Guide*, Dumpleton Software Consulting Pty Ltd, Parramatta, Australia, 1994.

[Freeman-Benson,1989] Bjorn N. Freeman-Benson, ‘‘Kaleidoscope: Mixing Objects, Constraints, and Imperative Programming,’’ *Proceedings of OOPSLA’89,* ACM Press, October, 1989, pp. 77-88.

[Garlan et al., 1995] David Garlan, Robert Allen, and John Ockerbloom, ‘‘Architectural mismatch: Why reuse is so hard,’’ *IEEE Software*, vo. 12, no. 6, November 1995, pp. 17-26

[Gabriel,1994] Richard P. Gabriel, ‘‘The Failure of Pattern Languages,’’ *Journal of Object Oriented Programming,* February 1994, pp. 84-88.

[Gamma et al., 1995] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA., 1995.

[Goguen,1986] Joseph A. Goguen, ‘‘Reusing and Interconnecting Reusable Components,’’ *Computer*, February 1986, pp. 16-28.

[Golberg & Robson, 1989] Adele Golberg and David Robson, *Smalltalk-80: The Language*, Addison-Wesley, Reading, MA., 1989.

[Guttag et al., 1985] John V. Guttag, James J. Horning, and Jeannette Wing, ‘‘An overview of the Larch family of specification languages,’’ *IEEE Software*, vol. 2, no. 5, September 1985, pp. 24-36.

[Hall & Weedon, 1993] Pat Hall and Ray Weedon, ‘‘Object-Oriented Module Interconnection Languages,’’ in *Advances in Software Reuse,* selected Papers from the Second Interna-

tional Workshop on Software Reusability, Lucca, Italy, March 24-26, 1993, IEEE Computer Society Press, pp. 29-38

[Helm et al., 1990] Richard Helm, Ian Holland, and D. Gangopadhyay, ‘‘Contracts: Specifying Behavioral Compositions in Object-Oriented Systems,’’ in *Proceedings of OOPSLA’90,* ACM Press, Ottawa, Canada, October 22-25, 1990.

[Johnson & Foote, 1988] Ralph E. Johnson and Brian Foote, ‘‘Designing Reusable Classes,’’ *Journal of Object-Oriented Programming,* August/September 1988.

[Johnson, 1992] Ralph E. Johnson, ‘‘Documenting Frameworks using Patterns,’’ *Proceedings of OOPSLA’92,* Vancouver, B.C., 18-22 October, 1992, ACM Press, pp. 63-76.

[Johnson, 1994] Ralph Johnson, ‘‘Why a Conference on Pattern Languages,’’ *Software Engineering Notes*, vol. 19, no. 1, January 1994, pp. 50-52.

[Kiczales & Lamping,1992] Gregor Kiczales and John Lamping, ‘‘Issues in the Design and Documentation of Class Libraries,’’ *Proceedings of OOPSLA’92,* October 1992, pp. 435-451.

[Lam & Shankar,1992] Simon S. Lam and Udaya Shankar, ‘‘Specifying Modules to Satisfy Interfaces: A State Transition Approach,’’ *Distributed Computing*, vol. 6, 1992, pp. 39-63.

[Linton et al., 1989] Mark Linton, John Vlissides, and Paul Calder, ‘‘Composing User Interfaces with Interviews,’’ *IEEE Computer*, Feb. 1989, pp. 8-22.

[Mili et al., 1990] Hafedh Mili, Johsn Sibert, and Yoav Intrator, ‘‘An Object-Oriented Model Based on Relations,’’ *Journal of Systems and Software*, vol. 12, May 1990, pp. 139-155.

[Mili et al., 1994a] Ali Mili, Rym Mili, and Roland Mittermeir, ‘‘Storing and Retrieving Software Components: A Refinement-Based Approach,’’ *Proceedings of the Sixteenth International Conference on Software Engineering*, Sorrento, Italy, May 1994.

[Mili et al., 1994b] Hafedh Mili, Odile Marcotte, and Anas Kabbaj, ‘‘Intelligent Component Retrieval for Software Reuse,’’ *Proceedings of the Third Maghrebian Conference on AI and SE*, April 1994, Rabat - Morocco, pp. 101-114.

[Mili et al., 1995] Hafedh Mili, Fatma Mili, and Ali Mili, ‘‘Reusing Software: Issues and Research Directions,’’ I*EEE Transactions on Software Engineering,* June 1995, vol. 21, no. 6, pp. 528-562.

[Mili, 1996] Hafedh Mili, ‘‘On Behavioral Descriptions in Object-Oriented Programming,’’ *Journal of Systems and Software*, August 1996.

[Mili et al., 1997] Hafedh Mili, Estelle Ah-Ki, Robert Godin, and Hamid Mcheick, ‘‘Another nail to the coffin of multi-faceted component classification and retrieval,’’ *Proceedings of 1997 Symposium on Software Reuse (SSR’97)*, May 1997, Boston Mass.

[Reenskaug, 1995] Trygve Reenskaug, *Working with Objects*, Prentice-Hall, 1995

[Saraswat,1989] Vijay A. Saraswat, *Concurrent Constraint Programming Languages*, Ph.D. thesis, Carnegie Mellon University, January 1989.

[Tracz,1993] Will Tracz, ‘‘LILEANNA: A Parameterized Programming Language,’’ in *Advances in Software Reuse,* selected Papers from the Second International Workshop on Software Reusability, Lucca, Italy, March 24-26, 1993, IEEE Computer Society Press, pp. 66-78.

[Wegner,1992]Peter Wegner, ‘‘Dimensions of Object-Oriented Modeling,’’ *COMPUTER* (special issue of on Object-Oriented Computing) IEEE CS Press, vol. 25, No 10, October 1992, pp. 12-20.

[Wilk, 1991] Michael Wilk, ‘‘Equate: An Object-Oriented Constraint Solver,’’ in *Proceedings of OOPSLA’91*, Phoenix, AZ., October 6-11, 1991, ACM Press, pp. 286-298.

[Yellin,1994] Daniel Yellin, Robert Strom, ‘‘Interfaces, Protocols, and the Semi-Automatic Construction of Software Adaptors,’’ *Proceedings of OOPSLA’94,* Portland, Oregon, October

1994, ACM Press, pp: 176-190.

[Zaremski & Wing, 1993] Amy M. Zaremski and Jeannette M. Wing, ‘‘Signature Matching: A Key to Reuse,’’ *Software Engineering Notes*, vol. 18, no. 5 (proceedings of the first ACM SIGSOFT Symposium on the Foundations of Software Engineering), pp. 182-190.